MUSS

USER MANUAL

VOLUME 2

This Manual gives the specification of the use of high-level languages.

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUE 12

DRAFT: October 2019

The MUSS User Manuals were published in three volumes:

```
MUSS   USER    MANUAL VOLUME 1:  Basic Facilities of MUSS

MUSS LANGUAGE  MANUAL VOLUME 2:  High-level Languages

MUSS UTILITIES MANUAL VOLUME 3:  Utilities
```

It may be helpful to read this Manual in conjunction with the MUSS User Manual Vol. 1 Issue 12, 1986 which can be found at archive.org .

This edition of Volume 2 was intended to be for Issue 12. However, it is an undated draft with several parts missing or incomplete. I have not found a more complete version.

This text was created by OCR conversion of the original, followed by extensive editing. For the most part the OCR was quite good. However, since in the original text "." and "," were very similar, this was a common error.

There was also a problem with use of quotes. I have not yet decided how best to handle this, so there is some inconsistency in the text.

Few typos were found in the original text and none were serious. Just a few "off by one" index errors, all quite obvious. I left the spelling checker set to UK English.

Any comments or errors found should be reported to me at:

michaeldgodfrey@gmail.com.

October 9, 2019

Contents

USER MANUAL i

DRAFT: October 2019 iii

Contents v

1 INTRODUCTION 1

2 FORTRAN 77 3
- 2.1 INTRODUCTION . . . 3
- 2.2 COMPILER ORGANISATION . . . 3
- 2.3 NON-STANDARD FEATURES . . . 3
  - 2.3.1 Holleriths . . . 3
  - 2.3.2 Fortran Character Set Extensions . . . 3
  - 2.3.3 MUSS Commands and Compiler Directives . . . 4
  - 2.3.4 Relaxation of Subprogram Argument Checks . . . 4
  - 2.3.5 Precision of Arithmetic Data Types . . . 5
  - 2.3.6 Data initialisation in Declarative Statements . . . 5
  - 2.3.7 Common Block Extensions . . . 5
  - 2.3.8 Length of Symbolic Names . . . 6
  - 2.3.9 Recursion of Subroutines and Functions . . . 6
  - 2.3.10 DO Loop Extensions . . . 6
  - 2.3.11 ENCODE and DECODE Statements . . . 7
  - 2.3.12 Direct Access READ and WRITE Industry-Standard Format . . . 9
  - 2.3.13 Octal and Hexadecimal Typeless Constants . . . 10
  - 2.3.14 Parenthesis in PARAMETER Statements . . . 11
- 2.4 COMPILER DIRECTIVES . . . 12
  - 2.4.1 *END – End of Program . . . 12
  - 2.4.2 *IMPORT – Import Library Procedures . . . 12
  - 2.4.3 *LIB - Import Whole Library . . . 12

2.4.4 *EXPORT Creating Fortran Libraries 13
2.4.5 *MAP Storage Control in Fortran 13
2.4.6 *LIST – Source Program Listing 15
2.4.7 *NOLIST – Stop Source Program Listing 15
2.4.8 *ANSISWITCH – Suppress Warning of Long Symbolic Names 16
2.5 COMPILER PARAMETERS 16
2.5.1 Compiler Options Parameter 16
2.6 INPUT/OUTPUT 20
2.7 FAULT MONITORING 21
2.7.1 Compile Time Faults 21
2.7.2 Run Time Faults 21
2.8 MULTI LANGUAGE PROGRAMMING CONSIDERATIONS 23
2.8.1 Variables as Dummy Arguments 23
2.8.2 Arrays as Dummy Arguments 23
2.8.3 Asterisks as Dummy Arguments 24
2.8.4 Procedures as Dummy Arguments 25
2.8.5 Complex Arguments and Functions 26
2.8.6 Character Functions 27
2.8.7 Parameter Conventions for Library Procedures Calls 27
2.9 INSTALLATION DEPENDENCE 27
2.9.1 Accuracy 27
2.9.1.1 MU6G 27
2.9.1.2 VAX 28
2.9.1.3 MC68000 28
2.9.2 Stop and Pause 28

3 PASCAL 29
3.1 INTRODUCTION 29
3.2 IMPLEMENTATION DEFINED RESTRICTIONS 29
3.3 EXTENSIONS 29
3.4 CONFORMANCE TO PASCAL TEST SUITE 31
3.5 INPUT AND OUTPUT 31
3.6 COMPILE TIME PRAGMATS 32
3.7 USE OF LIBRARY PROCEDURES 32
3.8 PASCAL LIBRARIES 34
3.9 RUN TIME STORE USE 35

4 BASIC 37
4.1 INTRODUCTION 37

4.2 COMPILER ORGANISATION 37
4.3 NON-STANDARD FEATURES 38
4.3.1 Standard Deficiencies in the Compiler 38
4.3.2 Character Set 38
4.3.3 Arithmetic Data Types 38
4.3.4 Line-number 41
4.3.5 MUSS Commands 41
4.4 COMPILER DIRECTIVES 41
4.4.1 *END – End of Program 42
4.5 COMPILER PARAMETERS 42
4.5.1 Compiler Options 42
4.6 IMPLEMENTATION - DEFINED FEATURES 42
4.7 INPUT/OUTPUT 49
4.8 MIXED LANGUAGE PROGRAMMING 49
4.9 ERROR HANDLING 50

5 MUSS C COMPILER **51**
5.1 INTRODUCTION 51
5.2 RESTRICTIONS AND EXTENSIONS 51
5.3 GENERAL COMPILER OPERATION 54
5.4 STORAGE CONTROL 56
5.5 IMPLEMENTATION AND MACHINE DEPENDENT INFORMATION 56
5.6 INTERFACING MUSS C PROGRAMS AND MUSS LIBRARIES 57

Index **59**

1 INTRODUCTION

CHAPTER 1 INTRODUCTION

2 FORTRAN 77

2.1 INTRODUCTION

The MUSS FORTRAN compiler implements Fortran 77 as defined in the document 'BSR X3.9 FORTRAN 77 dpANS FORTRAN Language X353/90' of the American National Standards Institute. The compiler attempts to remain close to the defined standard, any deviations from the standard are described in Section 2.3.

2.2 COMPILER ORGANISATION

2.3 NON-STANDARD FEATURES

MUSS Fortran includes extensions to the standard which improve the compatibility with ANSI-66 Fortran, improve efficiency on small machines by allowing a wider range of data types, and to assist in the transfer of Fortran software, albeit non-standard, to MUSS. Thus the more common extensions available in many of the other present day Fortran 77 compilation systems are supported.

All violations to the Fortran 77 standard produce a warning message, these messages may be suppressed by setting a compiler option.

2.3.1 Holleriths

Holleriths have been included as an extension to the standard language. The use of Hollerith constants is restricted to DATA statements in a manner described in Appendix C of the Fortran 77 standard. A Hollerith constant may only be used with INTEGER, LOGICAL or REAL types. The actual number of characters in such types is machine dependent (see Section 2.7).

2.3.2 Fortran Character Set Extensions

The standard Fortran character set has been extended to include lower case letters, the dollar sign '$', underscore '_', tab and EOT characters. During compilation lower case letters are accepted and treated as if they were upper case letters, everywhere except in character and Hollerith constants. Symbolic names may include dollar sign and underscore characters. A Fortran source line may include a tab character. A tab in columns 1 to 5 of the Fortran

source line has the affect of sufficient spaces to ensure the following character is treated as being in column 6. Tabs in column 6 and beyond are treated as spaces. The EOT character may be used to mark the end of the Fortran compilation and is an alternative to the *END directive (see Section 2.4) or the compiler reaching the end of the input document both of which indicate the end of the compilation.

2.3.3 MUSS Commands and Compiler Directives

MUSS commands may be included in the Fortran program source for optional interpretation at compile time as they are encountered. The option to interpret the MUSS commands is activated by the third Parameter to the compiler, as described in Section 2.8. MUSS commands to be interpreted at compile time consist of an asterisk in columns one and two of an input image followed by an alphabetic character which begins the command, any other format is treated as a comment image beginning with an asterisk and is ignored. If the option is not switched on the directives are treated as normal comments. MUSS commands can occur anywhere a Fortran comment could occur.

Compiler directives (which are described in Section 2.4) are also non-standard extensions to the Fortran standard language.

This compiler is more generous than the Standard in allowing any comments after the last END of the last program unit (and before any *END directive). Any program which has comments (or even MUSS commands) beyond the last END would be non-standard, but would be acceptable to this implementation.

2.3.4 Relaxation of Subprogram Argument Checks

The compiler has a non-standard option of relaxing the strict checking between dummy and actual arguments, of a subprogram. This option, when activated by setting a bit in the compiler mode parameter, permits an actual argument of REAL, INTEGER or LOGICAL type to be substituted for a dummy argument of either REAL, INTEGER or LOGICAL type, an actual argument of DOUBLE PRECISION or COMPLEX type to be substituted for a dummy argument of either DOUBLE PRECISION or COMPLEX type, and an actual argument which is an expression, variable, array element name or an array name to be substituted for a dummy argument which is a variable or an array.

2.3.5 Precision of Arithmetic Data Types

REAL is 32 bit precision, DOUBLE PRECISION 64 bit and COMPLEX components 32 bits. INTEGER and LOGICAL is either 16 bit or 32 bit precision, depending on the installation. The compiler also provides for INTEGER and LOGICAL data types with precision of 8, 16 and 32 bits.

In addition to the standard arithmetic data type words REAL, DOUBLE PRECISION, COMPLEX, INTEGER and LOGICAL the compiler is extended to recognise the following non-standard arithmetic data types.

```
INTEGER*1       8 bit integer
BYTE            Synonym for INTEGER*1
INTEGER*2       16 bit integer
INTEGER*4       32 bit integer
LOGICAL*1       8 bit logical
LOGICAL*2       16 bit logical
LOGICAL*4       32 bit logical
REAL*4          Synonym for REAL
REAL*8          Synonym for DOUBLE PRECISION
COMPLEX*8       Synonym for COMPLEX
```

The compiler is extended to allow these additional arithmetic data type words in any Fortran statement instead of a standard arithmetic data type word. Thus the compiler recognises additional data type words in:

type specification statements
implicit statements and
function statements.

2.3.6 Data initialisation in Declarative Statements

2.3.7 Common Block Extensions

There are several extensions concerning common:

i) The compiler is extended for initialisation of blank common. ANSI Fortran 77 only allows named common initialisation.

ii) The compiler is extended for blank common and named common to be initialised in any program unit. ANSI Fortran 77 only allows initialisation in block data program units.

iii) Character and non-character variables may be present in the same common block.

iv) Within separate program units the size of the same named common block can be different. However, after its initial definition, the size in a named common block declaration in a subsequent program unit may not be greater than in the initial definition.

2.3.8 Length of Symbolic Names

Symbolic names can have any number of characters.

2.3.9 Recursion of Subroutines and Functions

The compiler is extended to allow recursion of subroutines and functions. Normally all data variables are static, however non-common data variables may be allocated dynamically on the procedure call by the *MAP directive.

2.3.10 DO Loop Extensions

There are several extensions concerning DO loops:

i) In ANSI Fortran 77 if the iteration count of a DO loop is initially zero the statements in a DO loop are not obeyed. In Fortran 66 all DO loops were executed at least once. The compiler is extended for Fortran 66 loops, the compiler option (LOOP66) allows DO loops to be executed at least once.

ii) Fortran 66 has the notion of an extended DO loop range; ANSI Fortran 77 does not. The compiler option (EX-RANGE) allows a DO loop range to be extended.

2.3.11 ENCODE and DECODE Statements

These are additional statements which are an extension to ANSI Fortran 77.

The DECODE statement transfers data from variables or arrays in external form to variables or arrays in internal storage.

DECODE translates the data from character form to internal form according to a format identifier. The form of a DECODE statement is:

DECODE (c, f, b [, IOSTAT = ios] [, ERR = s]) [list]

c An integer expression that indicates the number of characters to be translated to internal form.

f A format identifier. It can be either the label of a FORMAT statement, or the name of a character array, character variable, character expression, or arithmetic array.

b The name of an array, array element, variable, or character substring reference. It contains the characters to be translated to internal form. The array or variable can be of any type.

ios An integer variable or integer array element that is defined as a positive integer if an error occurs, and as a zero if no error occurs.

s The label of an executable statement.

list A list of the data items which receive the data after translation to internal form.

The DECODE statement translates the character data in b to internal (binary) form according to the format specifier, and stores the elements in the list, as does a READ statement.

If b is an array, its elements are processed in the order of their subscripts.

The number of characters that the DECODE statement can process depends on the data type of b in that statement. For example, an INTEGER*2 array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array. A character variable or character array element can contain characters equal in number to its length. A character array can contain characters equal in number to the length of each element multiplied by the number of elements.

In a DECODE statement, c cannot be greater than b. If it is, the results are unpredictable.

The format specifier works the same way that it would for a formatted READ or WRITE statement.

Example:

```
      DIMENSION I(3)
      CHARACTER*9 X
      DATA I /'123456789'/
      DECODE (9, 100, X) I
100   FORMAT (I3)
```

This example translates the nine characters in X to integer form (specified by statement 100), and stores them in array I as follows:

```
I(1) = 123
I(2) = 456
I(3) = 789
```

The ENCODE statement transfers data from variables or arrays in internal storage to variables or arrays in external form.

ENCODE translates the data from internal form to character form according to a format identifier. The form of an ENCODE statement is:

ENCODE (c, f, b [, IOSTAT = ios] [, ERR = s]) [list]

c An integer expression. It indicates the number of characters (bytes) to be translated to character form.

f A format identifier. It can be either the label of a FORMAT statement, or the name of a character array, character variable, character expression or arithmetic array.

b The name of an array, array element, variable, or character substring reference. It receives the characters after translation to external form.

ios An integer variable or integer array element that is defined as a positive integer if an error occurs, and as a zero if no error occurs.

s The label of an executable statement.

list Contains the data to be translated to character form.

The ENCODE statement translates the list elements to character form according to the format specifier, and stores the characters in b, as does a WRITE statement. If fewer than c characters are transmitted, the remaining character positions are filled with spaces.

If b is an array, its elements are processed in the order of their subscripts.

The number of characters that the ENCODE statement can process depends on the data type of b in that statement. For example, an INTEGER*2 array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array. A character variable or character array element can contain characters equal in number to its length. A character array can contain characters equal in number to the length of each element multiplied by the number of elements.

In an ENCODE statement, c must not be greater than b. If it is the results are unpredictable.

The format specifier works the same way that it would for a formatted READ or WRITE statement.

Example:

```
      CHARACTER*3 A(3)
      DATA I /123456789/
      ENCODE (9, 100, A) I
  100 FORMAT (I3)
```

This example translates the value of X to character form and stores the characters in the character array A as follows:

```
A(1) = '123'
A(2) = '456'
A(3) = '789'
```

2.3.12 Direct Access READ and WRITE Industry-Standard Format

The compiler is extended for industry-standard formats for READ and WRITE statements to direct access files.

Thus as alternatives to

```
READ([UNIT=]v[,[FMT=]format], REC=record.no
    [,ERR=s][,END=s][,IOSTAT=ios])[input list]
 WRITE([UNIT=]v[,[FMT=]format], REC=record.no
    [,ERR=s][,IOSTAT=ios])[output list]
```

you can use

```
READ(u'record.no[,[FMT=]format]
      [,ERR=s][,END=s][,IOSTAT=ios])[input list]
 WRITE(u'record.no[,[FMT=]format)
      [,ERR=s][,IOSTAT=ios])[output list]
```

2.3.13 Octal and Hexadecimal Typeless Constants

Constants may be written in octal and hexadecimal notation. These are alternative ways to represent numeric constants. They may be used wherever numeric constants are allowed.

A typeless constant in octal notation is a string of octal digits enclosed by apostrophes and followed by the alphabetic character O. This constant has the following form:

```
                'd[d]...'O
  d is a digit in the range 0 to 7.
```

A typeless constant in hexadecimal notation is a string of hexadecimal digits enclosed by apostrophes and followed by the alphabetic character X. This constant has the following form:

```
                'd[d]...'X
d   is a digit in the rage 0 to 9, or a letter in the range
   A to F or a to f.
```

Leading zeros are ignored in typeless constants. Up to 64 bits (22 octal digits, 16 hexadecimal digits) may be specified.

Examples:

```
 Octal:        '12760'O
               '5'O
               '66666666'O

Hexadecimal:  'A2C12D45'X
```

```
'FEDCBA'X
'345'X
```

Octal and hexadecimal constants are typeless numeric constants. They assume data types based on the way they are used as follows:

i) When the constant is used with a binary operator, including the assignment operator, the constant assumes the data type of the other operand.

ii) When a specific data type (usually integer) is required, that type is assumed for the constant.

iii) When the constant is used as an actual argument, no data type is assumed; however, a length of four bytes is always used.

iv) When the constant is used in any other context, a 32 bit INTEGER data type is assumed.

Examples:

Statement	Data Type of Constant
INTEGER*2 ICOUNT	
REAL *8 DOUBLE	
ALPHA = '67B12'X	REAL*4
JCOUNT = ICOUNT + '676'O	INTEGER*2
DOUBLE = 'FFE7A1'X	REAL*8
IF (K .EQ. '123'O) GO TO 50	INTEGER*4
X = Y('16'O) + 6	INTEGER*4
CALL ABC ('D49C2'X)	none
IF ('AD89'X) 1,2,3	INTEGER*4
I = '6543'O - 'A39'X	INTEGER*4

An octal or hexadecimal constant actually specifies as much as 8 bytes of data. These constants are right justified, and zero filled to the left and truncated to the left where necessary. A warning results if any non-zero digits are truncated.

2.3.14 Parenthesis in PARAMETER Statements

In ANSI Fortran 77 the form of the PARAMETER statement is:

```
PARAMETER (p=e[, p=e] ...  )
```

The compiler is extended to allow the enclosing parentheses to be omitted. If the enclosing parenthesis are omitted and the variable is untyped, the type of the constant variable is defined by the type of the constant value.

2.4 COMPILER DIRECTIVES

Compiler directives are Fortran statements (and therefore start after column 6) which commence with an asterisk. They take effect at the point they occur during the compilation of the Fortran source program. As with all other Fortran statements they may be continued with continuation lines, only columns 7-72 are significant, and all spaces, blank lines and comments are ignored.

2.4.1 *END – End of Program

The *END directive marks a end of the Fortran compilation. It is an alternative to the end of input document being detected or an EOT character.

2.4.2 *IMPORT – Import Library Procedures

The *IMPORT directive is provided to enable Fortran programs to call subprograms from a Library. The directive has as its arguments a list of procedure names represented by Fortran character constants or Fortran names.

```
*IMPORT ‘CAPTION', ‘OUTHEX', ‘OPENFILE'
*IMPORT ‘GFNIN', ‘GFNOUT'
```

are examples of import directives. Note that as an imported procedure name may have more than six characters, a character constant is used. If the procedure names specified are not known to the system, then the Library containing them must be opened prior to the Fortran compilation. The procedures imported into a Fortran program can be called as if they were subprograms from the programs being complied. This directive must occur before any Fortran statements.

2.4.3 *LIB - Import Whole Library

The *LIB directive is provided to import all the procedures from a particular library. It is an equivalent to importing each of the procedures individually with a *IMPORT directive. The *LIB directive has one argument which is

a Fortran character constant containing the name of the file containing the library. The named library must not be already open, as this directive will open the library.

```
*LIB 'USR14:  TIMER'
```

is an example of the *us directive. This directive must occur before any Fortran statements.

2.4.4 *EXPORT Creating Fortran Libraries

The *EXPORT directive is provided to direct the compiler to include the named subprograms in the interface of a Fortran library, in addition to using this directive the mode parameter to the compiler must have the correct value (4) to create a library. The library will be placed in the file indicated by the second parameter to the Fortran compiler. The arguments to the *EXPORT directive are the Fortran names of the subprograms to be included in the library.

```
FORTRAN 0 LIB 4
      *EXPORT TEST
      SUBROUTINE TEST
      PRINT *,'TEST'
      END
      *END
```

is an example of creating a library called LIB containing the Fortran subroutine called TEST. The *EXPORT directive must occur before any Fortran statements.

2.4.5 *MAP Storage Control in Fortran

The *MAP directive is provided to give the Fortran programmer control of the location of code, common and local variables. This would be useful when several libraries share a common block, or if a program is very large. When this directive is not used Fortran handles the storage control automatically. The TL.SEG (MUSS User Manual Vol. 1) procedure must be called in conjunction with this directive. to create and specify the MUTL segment in which to place the data or code.

There are three forms of the *MAP directive, for mapping code, common

and local data.

The *MAP CODE form is used to specify the MUTL segment into which any following code is to be placed. This segment should already have been declared to MUTL by using the TL.SEG procedure. The TL.SEG may be called by using the ** command option. This directive has two arguments, the first specifies the MUTL segment number, the second (optional) parameter specifies the segment size in bytes.

```
FORTRAN 0  0  %8000
**TLSEG 3 %0 -1 -3 %C
     *MAP CODE 3
     I = 1
```

is an example of the *MAP CODE directive which would place the code for `'I = 1'` (and any following code) into MUTL segment 3.

The *MAP CODE directive may occur anywhere within a Fortran program.

The *MAP for common is used to specify the MUTL segment into which the named common is to be placed. This segment should have already been declared to MUTL. The *MAP directive for common may only occur before a program unit, it cannot occur within a program unit.

```
FORTRAN 0   0  %8000
**TLSEG 3   %0 -1 -3 %C
**TLSEG 4   123456 -1 -3 %C
      *MAP /NAME/ 3
      *MAP // 4, 123456
      SUBROUTINE ONE
      COMMON /NAME/ A,B,C
      COMMON D,E,F
      END
```

is an example of mapping the named common /NAME/ into MUTL segment 3 and blank common into segment 4 (of size 123456 bytes). If the common blocks are not mapped explicitly Fortran will choose store for them automatically.

The *MAP DATA directive is used to map local data variables and unmapped common (non blank) into a MUTL segment. The *MAP DATA directive has two arguments specifying the MUTL segment and its (optional)

size in bytes. This directive may only occur before a program unit. It cannot occur within a program unit. A MUTL segment number of zero as an argument causes the explicit mapping to be disabled, and allows Fortran to resume automatic store control of local data.

```
FORTRAN  0  0  %8000
**TLSEG  3  %0  -1  -3  %C
**TLSEG  4  %0  -1  -3  %C
       *MAP  /COM/  3
       *MAP  DATA  4
       SUBROUTINE  TWO
       COMMON /COM/ A,B,C
       COMMON /OTHER/ D,E,F
       INTEGER I,J,K
       I = J+K
         .
         .
         .
       END
```

is an example of mapping the common /COM/ onto MUTL segment 3, and the common /OTHER/ and the variable I, J, K onto MUTL segment 4. If the *MAP DATA is not used Fortran will allocate storage space automatically.

The use of mode bit 8 of the compiler also affects storage management. The setting of mode bit 8 to place items on the stack takes precedence over the *MAP DATA directive.

2.4.6 *LIST – Source Program Listing

The *LIST directive enables the listing of the source program during compilation. Comment lines of the program are not listed.

2.4.7 *NOLIST – Stop Source Program Listing

The *NOLIST directive stops the source program listing.

2.4.8 *ANSISWITCH – Suppress Warning of Long Symbolic Names

Normally the compiler issues a non-standard warning for Fortran names longer than six characters. The *ANSISWITCH turns the printing of this warning message on and off. The first occurrence of *ANSISWITCH suppresses the printing. Its second occurrence turns the printing on again, the third turns it off again, and so on.

2.5 COMPILER PARAMETERS

The compiler parameters were previously mentioned in the MUSS User Manual.

The first parameter specifies the origin of the Fortran 77 source.

The second parameter specifies the destination of the compiler output.

The third compiler parameter specifies the compilation options.

The fourth parameter specifies the maximum number of library interface procedures.

2.5.1 Compiler Options Parameter

Compiler options are specified as a list of keywords separated by commas. Some of the keywords may be followed by integers expressed in decimal or hexadecimal. No spaces are permitted in the list of options unless the whole list is enclosed in quotes.

Option keywords may be chosen from the following list (recognised abbreviations are in parentheses):

CLI	Specifies that any line with ** in columns one and two, followed by a letter in column 3, is a MUSS command language line to be interpreted. This is the default.
DEBUG(DB)	Specifies that debugging information is to be produced (the default).
ERRORLIMIT(EL)[number]	Changes the maximum of errors that can occur before the compiler abandons the compilation.

LIBRARY(LIB)	Specify that the source being complied is to be made into a library.
LIST(LS)	Lists the source as it is compiled until an optional *NOLIST directive is encountered. Comments are not listed.
LONGINTEGER(LI)	Specifies the default integer and logical size to be 32 bits. It does not affect those types sized explicitly. Any types sized explicitly would not conform to the ANSI standard for FORTRAN77.
LOOP66(L66)	Specifies that the code planted by the compiler for DO-loops and implied DO-loops should make the minimum number of times a loop is obeyed once, as in ANSI-66 FORTRAN.
LOOP77(L77)	Specifies that the code planted by the compiler for DO-loops and implied DO-loops should make the minimum number of times a loop is obeyed zero, as in standard FORTRAN77 (the default).
NOCLI	Specifies that any line with ** in columns one and two, followed by a letter in column 3, is not to be interpreted as a Command language line, but as a comment.
NODEBUG(NDB)	Specifies that debugging information is not to be produced.
NOLIST(NLS)	Do not list the source as it is compiled until an optional *LIST directive is encountered. This is the default.
NON−ANSI	Disables all warnings that indicate how the compiled program deviates from ANSI FORTRAN77. Each deviant feature can also be controlled individually to allow the programmer to be aware of any unexpected non-standard use. The options for the detailed manipulations of these warnings are given below.

ALTDA(AD)	Disables the warnings given when alternate forms of direct access READ and WRITE statements are used.
COMMON−SIZE(SZ)	Disables the warnings given when various references to the same COMMON block are of different sizes.
ENCODE.DECODE(ED)	Disables the warnings given when the ENCODE or DECODE statements are used.
EXTRACHARS(EC)	Disables the warning message given when the non-standard characters underscore '_' and dollar sign '$' are used in a symbolic name.
EXTRATYPES(ET)	Disables the warning given when types are given an explicit size, (e.g. INTEGER*4).
EX−RANGE(XR)	Disables the warnings given when attempting to use the extended range of a DO by jumping back into a DO-loop from outside.
HEX/OCTAL(XO)	Disables the warnings given when the hexadecimal or octal form of constants is used.
HOLLERITHS(HO)	Disables the warnings given when Hollerith constants are used as an extension to the ANSI standard, and allows arithmetic arrays containing Hollerith data to be used as format specifiers.
INIT−COMMON(IC)	Disables the warnings given when blank common is initialised or when a named common is initialised outside of a block data subprogram.
INIT−DECL(ID)	Disables the warnings given when variables are initialised outside of a block data subprogram.
LONGNAMES(LN)	Disables the warnings given for names longer than six characters. Names can be of any length.
MIX−CHAR(MC)	Disables the warnings given when CHARACTER variables are mixed with other types in COMMON or by EQUIVALENCE statements.

NOARGCHECK(NAC)	Disables warning messages given by the strict argument checking performed in the compiler between the definition and reference of program units complied at one time.
OLDPARAMETER(OD)	Disables the warnings given when a PARAMETER statement with no parentheses is used.
QUOTE−HOLL(QH)	Disables the warnings given when quotes are used to indicate Hollerith data.
RECURSION(RC)	Disables the warning message given when recursion is used.
NOTL	Specifies that the code generator should not be initialised.
NOTEND	Specifies that the code generator should not be terminated at the end of compilation.
number	This option is permitted to allow compatibility with earlier versions of the compiler, and to allow implementers to enable personalised options.
ONSTACK(OS)	Specifies that any variables not explicitly static (by being SAVEed or in common) are allocated storage on the stack,
OPENDEFAULT(OD)	Specifies that any OPEN statements within the program are not for both read and write. The file should be opened with only the access required by the first READ or WRITE statement that accesses it.
OPENDEFAULTIO(ODIO)	Specifies that any OPEN statements within the program should open the file with both read and write access allowed (the default).
SHORTINTEGER(SI)	Specifies the default integer and logical size to be 16 bits. It does not affect those types sized explicitly. Any types sized explicitly would not conform to the ANSI standard for FORTRAN77.

SYNTAXONLY(SO)	Specifies that only a syntax check should be performed on the program and no code generated.
TLPRINT<number>	This causes the compiler to issue a call with the number as a parameter to TL.PRINT (see MUTL Chapter in the MUSS User Manual) at the start of a compilation.

2.6 INPUT/OUTPUT

Input/output is implemented to the Fortran Standard. In extension to this standard characters may be read or written from REAL, INTEGER or LOGICAL data types using the 'A' format (See also Section 2.2).

The details of pre-connection are left undefined by the standard. In the implementation unit numbers are related to MUSS streams for the purpose of pre-connection. The result of unit modulus 8 gives the MUSS stream number which is used for the transput. It is the first access to the unit which defines if it is an input or output stream. However, some operations on a pre-connected unit do not imply input or output (namely BACKSPACE, REWIND, INQUIRE, OPEN, CLOSE). In these cases there should not be an input and output stream of the same number (unless it is an IO stream), and then it is unambiguous which stream should be used.

The BACKSPACE statement cannot be used to move between sections of a multi-sectioned MUSS stream. REWIND operates differently, rewinding a unit causes the stream to be ended and re-connected on the next access. This affects output to a process; any output destined for a process will be sent when a rewind is made, each rewind causing a further document to be sent.

It is not possible, at present, to BACKSPACE unformatted records, neither does the implementation allow the changing from reading to writing on pre-connected streams which are only defined for one mode of operation (an IO stream should be used instead).

When an INQUIRE by name is made to a pre-connected file, the inquiry will not be able to determine which unit the file is connected to, unless the unit has been accessed previously in the same run. If such an inquiry is made only details about the file will be returned, and not any about the connection. An OPEN statement which gives no file name causes a connection to the

MUSS current file. A PRINT statement or a READ with no unit number, or a '*' as a unit refers to input or output stream zero.

Output from programs consists of what was specified by the FORMAT, WRITE or PRINT statements. The carriage control characters in the first column are not interpreted or removed, this enables output to be read back by the same format that was used to write it (as the standard requires). To send any output to a printer the LIST command should be used. This has two parameters (the file to be printed, and the destination device), and it processes the file producing the correct interpretation of the carriage control characters for printing.

Fortran sequential input/output has a default maximum record length. In Standard Fortran there is no way of specifying the record length for sequential input/output so the subprogram FIO.SET.UNIT.REC.L is used. This subprogram has two parameters, the first the unit number and the second the maximum record length required. If this procedure is not used a suitable default is assumed. This procedure may be used for both program connected and pre-connected units.

2.7 FAULT MONITORING

2.7.1 Compile Time Faults

Faults at compile time are divided into warnings and fatal errors. A program containing only warnings may be run, as the warnings only refer to non-standard or bad features of the program (such as GOTO the next statement). Fatal errors are issued when the compiler is unable to understand the supplied program and may generate incomplete code for the offending statement. Each faulty line is output by the compiler followed by the fault messages.

Every fault message will attempt to locate within a statement by either an upward arrow below the point the compiler reached on determining the fault or including an offending name from the statement in the message. The former method is usually used for syntactic faults, the later for semantic faults. At the end of a compilation the total number of message faults is printed.

2.7.2 Run Time Faults

Faults at runtime will cause MUSS to trap the program, and these traps will be handled as any other high level language trap. A fault message usually

accompanies such a trap. The Fortran run-time system uses trap number 6 for any input/output faults. A list of Fortran trap 6 reasons is given below.

101 Inconsistent field descriptor for input/output list Item.
102 Illegal character in list directed complex character.
103 Illegal use of null value in list directed complex constant.
104 Attempted read beyond end of record.
105 No field descriptor for input/output list Item.
106 Illegal character in integer or exponent.
107 Illegal value separator.
108 Illegal use of repeat counts.
109 Zero repeat count not allowed.
110 As 104.
111 Illegal character in logical item.
112 Illegal character (in repeat count?).
113 * missing from a repeat count.
114 Illegal character in a real.
115 Illegal sign in integer or exponent.
116 Attempted write beyond end of record.
117 Illegal carriage control char on output.
118 Illegal run time format.
119 Format label specified not defined.
120 No digit following sign.
121 Reading beyond sequential ENDFILE record.
122 Illegal unit access.
123 Invalid parameter in OPEN.
124 Invalid parameter in CLOSE.
125 Writing Direct Access record of wrong length.
126 Writing beyond sequential record.
127 Invalid unit number.
128 Too many units connected.
129 Invalid Fortran file format.
130 Attempted use of unimplemented I/O feature.

Fortran also generates a trap Class 8 Reason 111 for an Assigned GOTO with a faulty argument specifying an invalid label. The mathematical functions library which can be called from a Fortran program also generates class 8 faults.

2.8 MULTI LANGUAGE PROGRAMMING CONSIDERATIONS

As Fortran supports calls to procedures in a previously complied library, then potentially procedure calls to any other high level language supported by MUSS are possible. However, there are difficulties due to contrasting language concepts. To aid multi-language programming in Fortran in this manner the procedural conventions of Fortran are described in terms of the systems implementation language MUSL.

Arguments to Fortran functions and subroutines are passed in a similar manner. Sections 2.8.1 through 2.8.6 detail conventions concerned with calling Fortran procedures. Section 2.8.7 is concerned with calling library procedures from Fortran.

2.8.1 Variables as Dummy Arguments

An actual argument is passed as a bounded reference to a variable of the same type and precision as the dummy argument. For character arguments the bound of the reference is the maximum length of the dummy argument.

Example

```
----FORTRAN----
SUBROUTINE FSUB(X)
REAL X
X = SQRT (X**3 + LOG10(X))
END
----MUSL----
REAL32 [1] MY;
FSUB (↑MY);
```

2.8.2 Arrays as Dummy Arguments

An actual argument is passed as a bounded reference to a vector whose elements are of the same type and precision as the dummy argument. For non-character arguments the bound of the reference gives the maximum permitted size of the dummy argument. For character arguments the bound of the reference gives the maximum numbers of characters accessible via the dummy argument, and an additional value of default INTEGER is always passed which gives the length of the character elements in the array. This

value is used when the length of the array elements of the dummy arguments are of assumed size.

Examples

```
      ----FORTRAN----
      SUBROUTINE FARR(XA,XB,I,J)
      REAL XA(100), XB(I,J)
      INTEGER I,J
      .....
      XA (K) = XB (L,M+1) + XB (L,M-1)
      .....
      END
      ----MUSL---
      REAL32[200] XA;
      REAL32[4000] XB;
      ....
      FARR (↑XA, ↑XB.20.20)
      ------------------
      ----FORTRAN---.
      SUBROUTINE PRVEC (CH,I)
      CHARACTER CH (I) * (*)
100   WRITE (6, 100) (CH)
      100 FORMAT (1X, A)
      END
      ----MUSL---
      $LO8 [20] VEC;
      DATAVEC MESS ($LO8)
      'MESSAGE 1'
      'MESSAGE 2'
      END
      PRVEC (↑VEC,4,5); ::  5 elements of 4 characters
      PRVEC (↑MESS,9,2); ::  2 elements of 9 characters
```

2.8.3 Asterisks as Dummy Arguments

In Fortran when the dummy argument is an asterisk the actual argument must be an alternate return specifier. No information is passed for asterisk dummy arguments; the result of a Fortran subroutine is always INTEGER and it contains a non-negative value. A result of zero or greater than the number of asterisks in the dummy argument list indicates a normal return,

otherwise control is returned to the alternate return label associated with this result value. Thus one indicates the first alternate return label, two the second. etc., in the actual argument list.

Examples

```
----FORTRAN----
....
CALL SELECT (Y,*100,*300,*500)
....
SUBROUTINE SELECT (X, *, *, *)
REAL X
IF X.LT.0) RETURN 1
IF X.GE.0.AND.LT.100) RETURN 2
IF X.GT.1E6) RETURN
RETURN 3
END
----MUSL----
REAL32 Y;
INTEGER I:
IF SELECT (Y) => I > 0 < 4 THEN
   ALTERNATIVE I-1 FROM
      BEGIN    ::  First alternate return
      ...
      END
      BEGIN    ::  Second alternate return
      ...
      END
      BEGIN    ::  Third alternate return
      ...
      END
   END
FI;
----------------------
```

2.8.4 Procedures as Dummy Arguments

Only the address of a procedure actual argument is passed. There is no dynamic check to ensure compatibility between the argument list of the dummy and actual procedure argument, the onus is on the programmer to check carefully use of this facility.

Examples

```
----FORTRAN----
...
ANS = INTEGR (SIN,0,0.5);
...
REAL FUNCTION INTEGR (FX,A,B)
REAL FX, A, B
EXTERNAL FX
...
...
Y = 0.5*(FX(A+I*STEP) + FX(A+(I+1)*STEP)
END
----MUSL----
PSPEC MSIN(ADDR[REAL32])/REAL32
PROC MSIN(X);
MSIN = SIN(x↑[0]);
END
...
REAL32 ANS;
REAL32 [1] A,B;
0 => A[0];
0.5 => A[1];
INTEGR(↑MSIN, ↑A, ↑B) => ANS
--------------------
```

2.8.5 Complex Arguments and Functions

Complex variables are defined as variables of user type with two REAL32 fields for the real and imaginary components. For a complex function after the argument list is passed an additional parameter is passed which is an unbounded reference to a complex variable for the function value.

Examples

```
----FORTRAN----
COMPLEX FUNCTION CXADD (CA,CD)
COMPLEX CA,CD
CXADD = CA + CD
END
----MUSL----
TYPE COMPLEX IS REAL32 R, I;
```

```
COMPLEX[1] A, B;
COMPLEX RES;
....
CXADD (↑A, ↑B, ↑RES);
-------------
```

2.8.6 Character Functions

For a character function after the argument list is passed one additional parameter is passed which is a bounded reference to a byte vector for the function value.

Examples

```
----FORTRAN----
CHARACTER*(*) FUNCTION CONCAT (STR1,STR2)
CHARACTER*(*) STR1, STR2
CONCAT = STR1//STR2
END
----MUSL----
$LO8[100] RES;
$LO8[20] A, B;
INTEGER I,J,K,L;
CONCAT (PART(↑A,I,J),PART(↑B,K,L),↑RES);
----------------
```

2.8.7 Parameter Conventions for Library Procedures Calls

2.9 INSTALLATION DEPENDENCE

2.9.1 Accuracy

2.9.1.1 MU6G

On MU6G INTEGER is in the range -2**31 to 2**31. REAL has a mantissa of 24 binary digits (approximately 7 decimal digits) with an exponent range 16**64 to 16**(-64) which gives a maximum decimal magnitude of 10**77 approximately.

DOUBLE PRECISION has an accuracy of 16 decimal digits approximately.

INTEGER, LOGICAL and REAL may hold up to four characters.

2.9.1.2 VAX

On VAX INTEGER is in the range -2**31 to 2**31. REAL has a mantissa of 24 binary digits (approximately 7 decimal digits) with an exponent range 2**64 to 2**(-64) which gives a maximum decimal magnitude of 10**19 approximately.

DOUBLE PRECISION has an accuracy of 16 decimal digits approximately.

INTEGER, LOGICAL and REAL items may hold up to four characters.

2.9.1.3 MC68000

On MC68000 INTEGER is in the range -2**15 to 2**15. REAL has a mantissa of 24 binary digits (approximately 7 decimal digits) with an exponent range of 2**64 to 2**(-64) which gives a maximum decimal magnitude of 10**19 approximately.

DOUBLE PRECISION has the same accuracy as REAL.

INTEGER and LOGICAL may hold up to two characters and REAL up to four characters.

The action of the EQUIVALENCE is non-standard. INTEGER and LOGICAL are allocated 2 bytes, REAL and DOUBLE PRECISION 4 bytes, and COMPLEX 8 bytes.

2.9.2 Stop and Pause

For STOP and PAUSE statements with no arguments, a caption indicating that the program has stopped or paused together with source program line number of the STOP or PAUSE statement is output on stream zero. When STOP and PAUSE have arguments the string of digits or characters is output with the caption.

After obeying a STOP or PAUSE control exits from the program to the caller. A paused program may be re-entered by the command CONTINUE.

3 PASCAL

3.1 INTRODUCTION

The Standard Pascal referred to here is as specified by British Standard BS6192 : 1982 Specification for Computer Programming Language Pascal. This is technically equivalent to ISO/DIS 7185. The full standard language is implemented but some implementation defined restrictions and extensions are listed below.

3.2 IMPLEMENTATION DEFINED RESTRICTIONS

1. Character strings are restricted to a maximum length of 140 characters.

2. Sets are restricted to 128 elements. Set types may be defined over any ordinal type with ordinal values in the range 0 to 127.

3. Run-time checking is restricted to detection of the following errors:

   (i) Subrange variable out of range.
   (ii) Non-existent CASE chosen.
   (iii) Illegal pointer access (only in simple cases).

4. Maxint is $2^{31} - 1$. Pascal uses 16-bit arithmetic where possible on 16-bit machines so declaration of variables as subranges rather than integers will improve code efficiency. If the N-pragmat is used (see Section 3.6 below) Maxint is restricted to $2^{15} - 1$ and only 16-bit arithmetic is used.

5. Real numbers are represented in 32-bits (except when the R+ pragmat is used).

6. Files are not permitted as components of arrays or records (in Issue 12).

3.3 EXTENSIONS

With the I+ option all deviations from the ISO standard are monitored but code is still complied for these extensions whenever possible, with I- the following extensions are permitted and not monitored.

1. Facilities are available to call any suitable MUSS library procedure. See Section 3.7 below. Additionally the following functions are provided

```
SIGN(X1,X2)
DIM(X1,X2)
LOG10(X)
ARCSIN(X)
ARCCOS(X)
SINH(X)
COSH(X)
TANH(X)
SIGN(X)
```

   where X's are either integer or real type. These each correspond to one or more of the MUTL mathematical functions.

2. An OTHERWISE option is provided for case statements where the case constants cannot all be listed. If the CASE expression is of integer (i.e. not subrange) type then an error will be registered rather than the OTHERWISE alternative being chosen if the expression is outside the range: LOW ... HIGH where LOW is the minimum of -1000 and the lowest CASE constant listed and HIGH is the maximum of +1000 and the highest CASE constant listed. Thus a wider range can be obtained by suitably choosing the CASE constants listed.

3. Hexadecimal constants are accepted. The hexadecimal digits must be preceded by % and be at most 32 bits.

4. Any line commencing with ** is interpreted as a MUSS command to be obeyed at compile time. The main use of this feature is to include source text from other files, or to direct the action of the MUTL system when compiling a library (see Section 3.8) or placing variables in a specific segment (see Section 3.9)

5. Renaming of files is permitted in the program parameters. (See Section 3.5.)

6. A limited facility for compiling independent modules is available. (See Section 3.8 below.)

3.4 CONFORMANCE TO PASCAL TEST SUITE

The compiler has been validated against the Pascal processor validation suite of B. A. Wichmann and A. H. J. Sale. Details of the results are in the PASCAL Implementation Manual Section 2.4.

3.5 INPUT AND OUTPUT

The files INPUT and OUTPUT are predefined to be the current MUSS input and output stream when the program run is started or the library opened. Character input is implemented in a way suitable for on-line operation on a line by line basis. Input is not requested until actually required by the program (so called lazy input).

All external files must be specified as parameters in the program heading corresponding to the MUSS files of the same name. The MUSS name will be upper case since Pascal treats upper and lower case identically. An extension allows parameters of the form Pascal filename = MUSS document where MUSS document is any of the ways of specifying a document described in Chapter 3 in the User Manual.

There is no limit as to the number of files a programmer may use but the maximum number of "readable" and of "writable" files is limited by the number of streams available in MUSS. Files are initialised to "readable" and "writable" by the procedures RESET and REWRITE. If the number of "readable"/"writable" files is insufficient then the procedure CLOSE(f) may be called to reduce the number of files in that state.

Pascal Text files and files of unstructured type correspond to unstructured MUSS files. In Text files explicit newline characters are included. It is possible to use MUSS to create a Text file directly in which case a newline character should appear before the end of file if Pascal is to process the file correctly.

Pascal files of structured types correspond to MUSS files of records.

The Pascal input and output system imposes an additional level on the basic MUSS system. MUSS and Pascal I/0 commands can be mixed but extreme caution is needed over stream selection, the effect of the Pascal buffer variable and the results of eoln and eof.

3.6 COMPILE TIME PRAGMATS

These are of the form (*$<letter>+*) to turn on feature <letter> and (*$<letter>-*) to turn this feature off.

The allowed letters are:

C	Compile full run-time checking (default -).
I	Check conformance to ISO standard (default +).
L	List source text (default -).
S <integer>	Direct compilation into new MUSS code area.
D <integer>	Direct variables declared beyond this point into new MUSS data area. (See Section 3.9 below).
T	Compile run-time line number trace (default -).
R	Make all real variables beyond this point double precision (default -).
N	Make maximum integer size 16-bit (default - i.e. 32-bit). This may only be used at the start of compilation.

3.7 USE OF LIBRARY PROCEDURES

Pascal can be used to call MUSS library procedures. Such procedures may be in the system library or in other libraries. The library procedures may have been written in Pascal (see Section 3.8 below), MUSL or any other suitable language. In the present MUSS library organisation the required libraries must be opened when the program is complied and when it is run.

In the Pascal program a procedure declaration or function declaration with an external directive may be given. this is required for ISO standard programs. In fact the declaration may be omitted unless the procedure has parameters of user defined type or procedural or functional parameters.

In Pascal the fullstops in MUSS procedure names must be omitted.

The correspondence between Pascal parameter types and MUSS parameter types is as follows:

PASCAL	MUSS
`char :`	`LOGICAL8`
`integer :`	`INTEGER, INTEGER32`

```
real :                                REAL32 (or REAL64 with R +)
user defined :                        equivalent user defined
var char :                            ADDR LOGICAL8
var integer :                         ADDR INTEGER32
var real :                            ADDR REAL32 (or ADDR REAL64)
var array of ?  :                     ADDR [ ?  ]
var packed array [1...n] of char :
                                      ADDR [LOGICAL8]
integer (with N+ option)              INTEGER16
var integer (with N+ option)          ADDR INTEGER16
```

In addition MUSS LOGICAL8, LOGICAL16, LOGICAL32 and INTEGER, INTEGER8, INTEGER16 and INTEGER32 value parameters are compatible with any Pascal ordinal type.

LOGICAL64 parameters do not have a direct equivalent but can be treated as set of 1 .. 64.

When a string is handed as a MUSS ADDR [LOGICAL8] it may be inconvenient to fix the string size by giving an external declaration with a particular value of n. If no external declaration is given strings of varying sizes may be passed. Furthermore literal strings of 2 or more characters may be substituted.

e.g. DELETE ('OLDFILE')

A Pascal string filled out with trailing spaces or nulls may be shortened using the library procedure PSTRTRUNC.

```
e.g.
       var str :  packed array [1 .. 10] of char;
       begin
       str :  = 'hello    ';
       caption (pstrtrunc (str));
```

alternatively ADDBOUND can be used to shorten to a known length:

```
caption (addbound( str, 5));
```

ADDR [ ] is *not* compatible with Pascal conformant array parameters.

An example of an external declaration with a procedural parameter is

```
procedure settrap (class :  integer, procedure trapproc
                  (p1, p2 :  integer)) external:
```

In this particular example the MUSS trapping mechanism requires that trapproc has an identical specification to the system procedure entertrap. Consequently p1 and p2 must be declared integer16 where integer16 = -32767 ... 32767 on machines with MUSS INTEGER as 16 bits.

When parameter types are compatible but not identical a compile time warning message is generated, however suitable code is complied wherever possible.

3.8 PASCAL LIBRARIES

MUSS Pascal supports a limited module facility and it is also possible to compile a set of Pascal procedures into a library.

A module is identical to a program except that the word "program" is replaced by "module" and all outer level procedures are exported for use by other modules. Global variables are imported. In a "program" outer level variables are exported and procedures may be imported.

A particular form of module is a MUSS library. To compile a library the Pascal compiler is called with third parameter = 4 (see Section 2.5.2 User Manual Volume 1). Outer level procedures are placed in the library directory.

Example of a Pascal library consisting of three procedures Q, R and S.

```
(*$I-*)
**TLSEG 0 0 %400000 -1 6 (Allocate suitable segment
as runtime address of library -- segment address is
machine dependent)
MODULE P;
PROCEDURE Q(P1 :  INTEGER);
BEGIN
WRITELN('PROC Q : P1 = ',P1)
END;
PROCEDURE R(C : CHAR);
BEGIN
WRITE(C)
END;
PROCEDURE S(VAR C : CHAR):
BEGIN
WRITE('PROC S : C = '):
```

```
R(C);
END;
BEGIN
END.
```

3.9 RUN TIME STORE USE

The Pascal stack segment for variables is the MUSS stack segment. For programs with large data structures this segment may be insufficient for all the variables. In this case some variables should be placed in another segment in the way described below.

The Pascal heap initially uses one segment. (currently segment %21, on the VAX.) If more space is needed it is allocated dynamically by automatic creation of additional segments. Heap space is recovered by dispose and reused.

A procedure ATTACH (SEG.NO, PTR) is available in the run-time library to set a pointer to the start of a given segment. In order to use this procedure the following declaration should be made in the calling program:

PROCEDURE ATTACH (P1:INTEGER; VAR P2: <required pointer type>);
EXTERNAL;

The present implementation creates full size records for variant records without attempting to optimise the space allocated. The layout of fields within records is controlled by MUTL, presently to the byte level. Packed does not, at present. attempt any further packing to the bit level.

Pascal uses MUTL to create space for variables in the current data area. Variable declarations can be directed into other segments of store using **TL.SEG, **TLLOAD. and **TLDATAAREA as described in Section 18.4 of the MUSS User Manual Volume 1 e.g.

```
PROGRAM P;
        **TLSEG 2 0 %40000 -2 12
        **TLLOAD 2 3
        **TLDATAAREA 3
        VAR X,Y,Z: INTEGER;
        **TLDATAAREA 0
        a, b, c:  real;
```

puts the variables X, Y, Z in a segment 4 (on a VAX) and a, b, c on the

stack. Note that Pascal uses TLSEG 1 for the heap so this must not be reused. The D pragmat may be used instead of **TLDATAAREA.

4 BASIC

4.1 INTRODUCTION

The MUSS BASIC compiler implements BASIC as defined in the document 'Draft Proposed American National Standard for BASIC. X3J2/82-17'. Copies of this document are available for $20 from:

X3 Secretariat
CBEMA
311 First St., NW
Suite 500
Washington, DC 2001.

The compiler attempts to remain close to the defined standard, any deviations from standard are mentioned in Section 4.3. MUSS BASIC is available for the VAX, MC68000 and MU6G computers.

In Issue 12 the compiler attempts to implement the core module as defined in the standard (Chapter 4 through 12). In subsequent Issues it is intended to upgrade the compiler to implement the following modules:

Both enhanced file modules (Chapter 11), Graphics module (Chapter 13). This Chapter of the standard is under revision, and a large number of changes are anticipated. Thus the revised standard is awaited before this module is implemented.
Editing module (Chapter 16).

There are no immediate plans for the real-time and fixed decimal modules.

4.2 COMPILER ORGANISATION

The BASIC compiler complies a complete BASIC program. The issued BASIC compiler operates in a one pass manner to generate executable binary as an object program or a MUSS library. It is a straightforward task to produce a BASIC compiler that operates in a two pass manner, where the MUTL output from the first pass is the input of the second pass.

Examples

```
BASIC  SOURCE  BIN       ::  One pass
```

```
BASIC  SOURCE  MUBLOUT   ::  Two pass
LOAD   MUBLOUT BIN
```

4.3 NON-STANDARD FEATURES

MUSS BASIC includes extensions to the standard which improve efficiency on small machines by allowing a wider range of data types. Compiler directives are an extension to the standard. These are described in Section 4.4. The formal syntax productions are given for extensions. The syntax productions that replace those in the standard are highlighted with an '*'.

4.3.1 Standard Deficiencies in the Compiler

The following standard features are not implemented in Issue 12.

chain-statement
program-name-line
debug-statement
trace-statement
break-statement
erase-statement
OUTIN access mode
SKIP REST option in read-statement and input-statement.

4.3.2 Character Set

The definition of character is extended for tab characters.

Syntax extension

*unquoted-string-character=space/tab/plain-string-character

Semantics

A tab character is processed by the compiler as if it was a space character.

4.3.3 Arithmetic Data Types

The NUMERIC data type is either of 32 bits or 64 bits of real precision. The compiler also permits non-standard integer data types of precisions of 8, 16 and 32 bits. Additional keywords are as follows:

WORD integer or 16 bit precision

LONG integer of 32 bit precision

INTEGER integer of default precision, which is either 16 or 32 bit precision

REAL Synonym for NUMERIC.

The associated extensions to the standard are:

i) The declare-statement is extended to declare scalar and array variables of the data types.

Syntax extension

numeric-type > `(WORD/LONG/INTEGER/REAL)`
`(numeric-identifier/numeric-array-declaration)`
`(COMMA/WORD/LONG/INTEGER/REAL)?`
`(numeric-identifier/numeric-array-declaration))*`

Examples

```
DECLARE LONG I,J,K
DECLARE WORD 13,AI(1 TO 20, 0 TO 3)
DECLARE WORD W1,W2,LONG L1,L2,REAL R1,R2,R3(100)
```

Semantics

After a data type keyword in the declare statement all variables are of this type until another data type keyword is encountered.

ii) The declare statement is extended for the type of a function to be one of the extended numeric data types.

Syntax extension

*function-type = (WORD/LONG/INTEGER/REAL)?function-list
*function-list = defined-function(comma(WORD/LONG/INTEGER/REAL)?
defined-function)*

A data type keyword is only permitted when the defined-function is a numeric defined function.

Examples

```
DECLARE LONG FUNCTION LP1, LP2
DECLARE REAL FUNCTION RP1, RP2, WORD WP1, WP2
```

```
DECLARE FUNCTION      NUMF, STRPS
```

Semantics

After a data type keyword all functions are of that data type until another data type keyword is encountered.

iii) The function-definition is extended for the type of a function to be one of the extended numeric data types.

Syntax extension

*numeric-defined-function = (WORD/LONG/INTEGER/REAL)?numeric identifier

Examples

```
101 DEF LONG DFL(A,B) = A**2 - B**2 + K
200 FUNCTION WORD (X,Y,S$,I)
```

iv) Unless a numeric identifier is explicitly declared it acquires an implicit data type of real. The definition of numeric identifiers is extended for an implicit data type of integer. The precision of implicit integer is that of INTEGER.

Syntax extension

numeric-identifier = letter identifier-character percent-sign?

Examples

```
FRED
int-Sum
FN.FACTORIAL
```

v) The definition of numeric-constant is extended for integer constants.

Syntax extensions

```
*numeric-constant = sign?(numeric-rep/integer-rep)
integer-rep       = digit digit* percent-sign
```

Examples

```
1
10234
```

Semantics

Integer constants always have a percent sign. The precision of integer constants is INTEGER.

4.3.4 Line-number

The definition of a line-number has been extended to five digits.

Syntax extension

line-number = digit digit? digit? digit? digit?

4.3.5 MUSS Commands

The compiler is extended to allow MUSS commands to be included in the BASIC program source for interpretation at compile time as they are encountered. Whenever a source program line commences with ** the MUSS command interpreter is invoked to interpret the command following the **. Spaces and tab characters may precede the **.

Syntax extension

```
line > MUSS-command-line
MUSS-command-line = **MUSS-command
```

Examples

```
**IN SOURCEFILE
```

4.4 COMPILER DIRECTIVES

Compiler directives are written as a 'line' of a BASIC program. Normally these directives may appear before or after any other line of a BASIC source program. Compiler directives take effect at the point they occur during the compilation of a BASIC source program. A directive line does not have a line number but it may have a tail comment: all directives start with an asterisk followed by an alphabetic letter.

Syntax extension

line > compiler-directive-line

The compiler accepts the following compiler directives as non-standard extensions:

*END

4.4.1 *END – End of Program

A BASIC compilation ends when the end of the input source document is read. Alternatively the *END directive is used to mark the end of the BASIC program. This directive is needed whenever the program is input to the compiler from a terminal a line at a time.

Syntax extension

compiler-directive-line > *END tail

Examples

```
*END            | End of program – nonstandard.
```

4.5 COMPILER PARAMETERS

The compiler parameters where previously mentioned in the MUSS User Manual.

The first parameter specifies the origin of the BASIC source.

The second parameter defines the destination of the compiler output.

The third parameter specifies the compilation options.

The fourth parameter specifies the maximum number of library interface procedures.

4.5.1 Compiler Options

Bit 0 indicates that debugging is not to be produced.
Bit 2 indicates that the compilation is to produce a MUSS library.
Bit 8 indicates that INTEGER precision is 16 bits.
Bit 9 indicates that INTEGER precision is 32 bits.
Bit 10 indicates that numeric precision of numerics is 32 bits of real.
Bit 11 indicates that numeric precision of numerics is 64 bits of real.

4.6 IMPLEMENTATION - DEFINED FEATURES

This section details the implementation defined features of MUSS BASIC as described in Appendix 3 of the standard. Any differences between implementation defined features of MUSS BASIC on the VAX, MC68000 and MU6G are also detailed.

Section 2.3

– Interpretation of syntactically illegal constructs
syntax faults are reported to the user at compile time, and the compiler proceeds to the next line of the source program.

– format of error messages
A descriptive message indicating the nature of the fault detected is issued, and the compiler attempts to give an indication of the position of the offending item(s) in the source line.

Section 2.4

– format of exception messages
Exception messages are issued in one of two formats; as illustrated in the examples below;
"BASIC Exception 8002 : Too few data in input-reply"
"BASIC implementation – defined exception 7107 : Unable to open file".

– hardware dependent exceptions

– order of exception detection in a line

Section 4.1

– other-character
No other-characters are defined.

– coding for the native collating sequence.
The native collating sequence is the same as the standard collating sequence.

Section 4.2

– end-of-line
End-of-line is marked by the linefeed (%A) character.

– effect of parameter list in program-name-line of program

– relationship of program-designators and program-name

Section 5.1

– precision and range of numeric-constants

Numeric constants may be of the following types:
INTEGER 8-bit

INTEGER 16-bit
INTEGER 32-bit
REAL 32-bit
REAL 64-bit
On all machines INTEGER8 in the range -2^7 to 2^7. Similarly INTEGER16 is in the range -2^{15} to 2^{15}. INTEGER32 is in the range -2^{31} to 2^{31}.

1. MU6G
On MU6G, the default INTEGER is 32-bit. Default REAL is REAL 32-bit with a mantissa of 24 binary digits (approximately 7 decimal digits), with an exponent range 16^{64} to 16^{-64}, which gives a maximum decimal magnitude of 10^{77} approximately. REAL64 has an accuracy of 16 decimal digits approximately.

2. VAX
On VAX, the default integer is 32-bit. Default REAL is REAL 32-bit with a mantissa of 24 binary digits (approximately 7 decimal digits), with an exponent range 2^{64} to 2^{-64} which gives a maximum decimal magnitude of 10^{19} approximately. REAL64 has an accuracy of 16 decimal digits approximately.

3. MOTOROLA based machines
Default INTEGER is INTEGER16. Default REAL is REAL32 with a mantissa of 24 binary digits (approximately 7 decimal digits), which gives a maximum exponent of 10^{19} approximately. REAL64 has the same accuracy as default REAL.

Section 5.2

– Initial value of numeric variables
No initial value is defined.

Section 5.3

– order of evaluation of numeric-expressions
The user should not place any reliance on the order of evaluation.

Section 5.4

– accuracy of evaluation of numeric functions

– pseudo-random number sequence

– value of INF and EPS

– availability of calendar and clock

Both calendar and clock are provided as part of the MUSS BASIC System.

Section 5.6

– precision and range of numeric values as described in Section 5.2.

– precision and range of floating decimal arithmetic

– precision and range of native arithmetic

– accuracy of evaluation of numeric expressions

Section 6.2

– maximum length of undeclared string-variables

in the current Issue, undeclared string variables may have a maximum length of 255 characters.

– initial value of string variables

No initial value is defined.

Section 6.4

– values of CHR$ for the native character set

– values of ORD for the native character set

The native character set is defined to be the same as the standard character set.

–availability of calendar and clock

Both calendar and clock are provided as part of the MUSS BASIC System.

Section 6.6

– collating sequence under OPTIONS COLLATE NATIVE

This is defined to be the same as the standard collating sequence.

Section 7.2

– value of DET before INV has been invoked

No value is defined

– value of the inverse of a singular matrix

bf Section 9.1

– value of a defined function when no value has been specified

– initial values of local variables in external functions
No initial value is defined.

Section 9.2

– effect of redimensioning an array parameter when an element of that array is also a parameter

– initial values of variables which are not formal parameters to a procedure
No initial values are defined.

Section 9.3

– interpretation of the program-designator in a chain-statement

– initial values of variables in a chained-to program

– interpretation of upper-and lower-case letters in a program-designator

Section 10.2

– input prompt
The default input prompt is "→".

– means of requesting input in batch mode
Data to be supplied as input to a program to be executed off-line should be placed immediately after the "RUN" command on the job control stream.

Section 10.3

– significance width for printing numeric representations
6 for REAL32 values. 12 for REAL64 values.

– exrad width for printing numeric representations
The exrad width is 2 characters.

– effect of non-printing characters on columnar position
The columnar position is incremented for all characters generated during output processing.

– default zonewidth
The default zonewidth is 20 character positions.

– default margin
The default margin is 80 character positions.

– treatment of trailing space at end of print line
If generation of a trailing space during output of a numeric value would cause the margin to be exceeded by more than one, it is suppressed.

– effect of invoking a function which causes printing while printing
If the printing is to be to different channels, then the result of each print operation is as normally defined by the Standard. If, however, the same channel is specified for printing, the user should place no reliance of the format of the output generated on that channel.

– use of upper or lower case "E" in exrad
Upper case "E" is printed in an exrad.

– attempting to reopen a file under a different ARITHMETIC option

– two program-units attempting to open a file under different attributes or options

– means of insuring preservation of file contents between runs
This is achieved using the standard MUSS file system.

– effect of certain combinations of file organisation and type

– length of records in INTERNAL and NATIVE files
There is no limit placed on record length for these file organisations.

– maximum length of records when not specified or available

– value of DATUM for a non-stream file
The value is "UNKNOWN".

– value of ask-attribute NAME for channel zero
The value is "STR0*".

Section 11.0

– effect of certain combinations of file organisation and type
A file name whose last character is "*" is treated as the name of a process.

Section 11.1

– maximum channel number
Channel numbers may be in the range 0 to 255.

– whether a file name with different case letters denotes the same file or different files
Such names denote different files.

– effect of attempting to open an already open file

– number of channels which may be active simultaneously
The maximum number is eight.

– attempting to open a file with attributes different from those under which it was created
BASIC Exception 7106 "incorrect type of document" is generated.

– meaning of exception codes 7101-7199
7101 No streams available
7102
7103 Illegal stream name
7104 Stream undefined
7105 Incorrect type of stream
7106 Incorrect type of document
7107 Unable to open file
7110 Illegal combination of file attributes
7111 Keyed files not implemented
7112 Illegal key relation
7113 Incorrect mode for channel
7114 Attempted use of unimplemented I/O
7115 Inactive channel in RESET
7116 No operation since OPEN in channel.

Section 11.2

– method of signifying that data is not available for input on a non-file device channel

Section 11.3

– means of indicating end of record (EOR)
For files with DISPLAY organisation, EOR is indicated by a newline symbol (A). For INTERNAL files, EOR is indicated by a byte containing FF.

– accuracy of printed numeric values produced by PRINT for DISPLAY files
As defined for Section 10.3

Section 11.4

– retrieving a record from a NATIVE file having contents which are incompatible with the TEMPLATE

Section 11.5

– effect of data modification statements on files that are not RELATIVE or KEYED

– use of SKIP in incompatible template for REWRITE

Section 12.1

– value of EXTYPE for locally defined exceptions

Section 13.3

Module not implemented.

Section 14.6

Module not implemented.

Section 15.1

Module not implemented.

Section 16.0

Module not implemented.

4.7 INPUT/OUTPUT

To be completed.

4.8 MIXED LANGUAGE PROGRAMMING

Providing a compiler supports references to procedures in a previously complied library, then potentially procedures within libraries written in any other high level language supported by MUSS may be called from that language. However, there are difficulties due to individual language conventions. To assist in mixed language usage of BASIC, the parameter

conventions of BASIC procedures are described in terms of the systems implementation language MUSL.

To be completed.

4.9 ERROR HANDLING

To be completed.

5 MUSS C COMPILER

5.1 INTRODUCTION

The MUSS C compiler implements the C language as described in *The C Programming Language* by B. Kernighan and D. Ritchie (Prentice-Hall) except for the restrictions and extensions outlined below, which apply to the release of C in MUSS Issue 12. The compiler generates code through calls on the MUTL (Target Language) procedures and the code generated is intended to run in the MUSS environment. The set of files making up a C program can be considered in this context to be analogous to the set of modules constituting a MUSL program. and most of the information about the overall compiling scheme for MUSL is also relevant to this compiler.

5.2 RESTRICTIONS AND EXTENSIONS

1) The statement following the *switch* statement must be a block.

2) *Enumerated* types are not implemented.

3) Aggregate types (*structures* and *unions*) may be copied, compared, passed as parameters or returned as the results of functions.

4) *Structures* (not *auto*) containing fields may be initialised by giving a list of values for named members or fields, any unnamed fields will be set to zero.

5) *Structures* may not contain unnamed trailing fields for padding purposes.

6) Arrays of characters(*extern* or *static*) can be initialised by the elements of a string. If the initialiser concerned is enclosed in '(' ')', further elements can be given, (separated by ','), otherwise if the size of the array is known the remaining elements will be cleared to zero.

```
e.g.
    char a[10] = "abc";
    char b[20] = (4,"abc",5,"defy");
are permitted
    char c[10] = 4,"abc",5,6,7;
is not
```

7) Forward references to structure types are permitted but only where they are used in a context that does not require information about the actual storage size of the type. Generally speaking. this means that only instances of pointers to such types are acceptable.

8) Identifiers may contain up to 64 characters and all of these have significance in distinguishing between identifiers, not merely the first eight.

9) Functions which are called with a variable number of arguments must give a declaration at external definition level which specifies the maximum number of arguments with which the function will be called (and its result type if required). The syntax of this is almost identical in form to that for external definitions or declarations except that within the '(' and ')' specifying the identifier to be a function there appears an integer constant giving the maximum number of arguments.

```
e.g.
   extern char * sstrprnt(9);
   char (*x1(4))[];
```

For all functions declared implicitly by use in a function call context the maximum/formal number of arguments for the implicit declarations will be defined by the number of actual arguments at this first call of the function.

10 The compiler contains no integral source preprocessor. Any source submitted to the compiler must have been already 'preprocessed'. However the following compiler control lines are permitted

```
   #line <CONST.EXPR> <NAME>
```

```
Currently has no effect.
```

```
   #inform <CONST.EXPR>
```

This directive is introduced to control the printing of different kinds of information during compiling. (some information is mainly relevant to compiler debugging). The integer constant is bit encoded as follows:

```
Bit     Function
0       Print the encoded itemised line in hexadecimal
1       Print the source characters as read
3       Inhibit printing of list of imported library
          functions at end of each file
4       Print analysis record for expressions at end
of
          syntactic recognition and at end of semantic
          checking
5       Inhibit compiler generated compile map
6       Inhibit compiler generated warnings for
          members/fields which do not correspond to
          the structure pointer being accessed.
8-15    Used to control MUTL output i.e. they are passed
          as bits 0-7 of the parameter of TL.PRINT
          (described elsewhere), but the most usual
          bits to set in this directive are bit 10
          (MUTL prints a procedure address map) and
          bit 13 (MUTL prints data mapping
          information).
```

e.g. to print MUTL procedure map and MUTL data mapping information and to inhibit compiler generated compile map

```
#inform 0x2420

#end
```

This directive may be used to indicate the end of the source to the compiler. It is required if the source is not a file or is a file containing information following the program source.

11) In addition to the above, if a statement commences with the sequence '#**' then a call on any MUSS library procedure may be forced from the compiler using the line following the '#**' as a command to be processed by MUSS. This may typically be used to control the selection of different streams of input or to make direct calls on the MUTL procedures in order to control the environment of a compilation. For example to include a file at any point in compilation by using the MUSS 'IN' command.

#**IN FILENAME

12) Where a name in a function call context has no current declaration and matches a MUSS library name in a currently open library then the function call will be to this MUSS library procedure and the actual arguments must match the number and types of the formal parameters of the procedure concerned. Thus if any ambiguity of function identifier could occur a declaration may need to be made for functions declared later in the file, (the compiler prints a list of imported library functions at the end of each file). Alternatively the automatic linking into the MUSS library structure may be completely inhibited by setting a compiler mode bit (Section 2.5.2 User Manual Volume 1).

5.3 GENERAL COMPILER OPERATION

The MUSS C compiler operates in a similar manner to the MUSL compiler and it would be useful to the user to have read about the compiling scheme of MUSL as described in the opening paragraphs of the section describing the MUSL language in Volume 1. The compiler has the standard MUSS compiler format as described in Volume 1 Section 2.5.2.

The compiler operates on a statement by statement basis, but some statements may be broken down into smaller lexical units in order to avoid excessive sizes for various compiler lists. Thus if a line is echoed by the compiler the actual output may only be a part of the physical line presented to the compiler.

Where errors are detected the compiler will print the line number where

the fault occurred, a message telling what the fault was and the source line causing the fault with the symbols '<?' inserted to show the point in the line at which the error was detected.

e.g. 3. 174 >>>>*SYNTAX? |=int<?|;

(If the fault is a warning rather than a 'fatal' error a '>>>>?' message will be printed).

In addition the compiler will generate error messages when identifiers for labels, structures or functions are found to be undefined at the end of function bodies or of files.

In this release of C, all programs are complied to be a library either by setting the compiler mode bit appropriately or by the compiler. When the compiler mode bit is set all functions at external definition level are put into the library interface. If it is not set only the function main will be put in the library interface. To run the program the library must be opened and the function *main* called explicitly, (obviously any other function could also be called). Thus a simple compile sequence might look like:–

```
C O LIBNAME %5
main () (     /*this source could be a file */
CAPTION (*simple program*); /*calls on MUSS library procedures*/
NEWLINES(1);
)
#end      /*required if source not on a file*/
```

and to run this program the command sequence would be

```
LIB LIBNAME
main
```

If no output filename is given as the second parameter to C then the default 'a.out' will be substituted.

The following will compile a program consisting of three files and using a library CIO in the directory UTL with the output destination the library "a.out".

```
LIB UTIL:CIO
C file1.c  a.out %405 /*C file1.c 0 %401 has same effect*/
C file2.c  0 %C04
C file3.c  0 %804
```

Note that the MUTL mode bit indicating a library compile (%4) must be explicitly given for each file when compiling a library. The library could then be used by

```
LIB UTIL:CIO
LIB a.out
main
```

(Any parameters to *main* might need to be supplied to calling it from another, possibly MUSL, program) .

5.4 STORAGE CONTROL

At compiler initialisation time segments are created for the code of the program and for the *external* and *static* variables of the program. For the majority of programs it is assumed that these default actions will suffice. (It should be noted that initialisation of *extern* and *static* variables is performed at compile time and therefore conceptually only occurs as the library comprising a program is opened. Thus if the *main* procedure of an opened MUSS C program library were to be called twice the correct initialisation values would not be set).

If explicit storage control is required then the procedures TLSEG, TLLOAD and TLCODEAREA should be used (via the #** mechanism). The compiler uses MUTL data area number 2 for initialised *extern* and *static* variables, this is mapped to MUTL segment 1 which may be redefined to be of a different size if the default size of a segment is not suitable. For uninitialised *extern* and *static* variables MUTL data area number 3, mapped to MUTL segment 2, is used.

5.5 IMPLEMENTATION AND MACHINE DEPENDENT INFORMATION

For specific implementation information the MUSS C implementation Manual must be read, some more general information is given here.

1) The keyword *register* has no effect

2) *chars* are considered to be signed quantities (*unsigned char* may be used).

3) Parameters of MUSS C functions will occupy a size not necessarily consistent with the size of the type of the formal parameters and may not appear within the address space in the order in which they are formally declared. (Generally speaking parameters all occupy 8 bytes). Thus before accessing consecutive parameters using pointers a thorough understanding of the stack layout would be required.

4) Actual parameters are evaluated left to right.

5) Actual parameters which are of aggregate types are passed as pointers to the aggregate and copied to a local aggregate at function entry time.

6) Fields are allocated from the least significant end of the integer sized quantities within which they are packed and are considered to be unsigned quantities.

7) The default sizes of various types of variable are as follows for particular machines.

char, unsigned char	`8 bits (all machines)`
short, unsigned short	`16 bits (all machines)`
int, unsigned	`32 bits (all machines)`
long, unsigned long	`32 bits (all machines)`
float	`32 bits (all machines)`
double	`64 bits (all machines).`

5.6 INTERFACING MUSS C PROGRAMS AND MUSS LIBRARIES

Calls of C functions in C programs may have a variable number of arguments and the types of the arguments may vary and need not match that of the formal arguments.

For these reasons the arguments of C procedures are treated as being of MUTL type Logical64 (Section 18.3 User Manual Volume 1), this size of argument being sufficient to contain any of the 'basic' or pointer types.

The compiler does no type checking on these types of arguments (although expressions of type char or short are converted to int and those of type float to double). The compiler also permits the usage of MUSS library procedures as described earlier. In this case some of the formal arguments may be of explicit type and this may be checked by the compiler, (casts can be used to convert unsuitably typed expressions).

One restriction imposed by the MUSS C compiler is that pointers to ordinary MUTL procedures cannot be generated. It is assumed within the compiler that all pointers to functions are of the variable and fixed size argument type. If a pointer to a MUSS procedure is required it will be necessary to use an intermediary MUSS C function.

```
e.g.
    my.sin(x)
    double x:
    return SIN((float) x):          /*MUSS SIN*/
```

Also, it should be noted that parameters of aggregate types are passed to 'C' type procedures as pointers rather than the actual value.

Index

ANSI Fortran 66, 6
ANSI Fortran 77, 6

Basic: Introduction, 37

C, 51

Fortran 77, 3

MUSS BASIC, 37
MUSS C, 51
MUSS Fortran, 3
MUSS Pascal, 29
MUTL, 30

Pascal, 29